UNIVERSITAT POLITECNICA DE CATALUNYA

PARALLELISM

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

15th May 2019, Q2

Contents

1 Introduction **2**

2 Anlysis with *Tareador* **2**

2.1 Merge Function 2
2.2 Multisort Function 2
2.3 Task dependency graph (TDG) 3
2.4 Parallel Performance and Scalability 5

3 Parallelization and performance analysis with tasks **6**

3.1 Leaf Strategy 6
3.2 Tree Strategy 9
3.3 Task Cut–Off Mechanism 12

4 Parallelization and performance analysis with dependent tasks **16**

5 Annex **19**

5.1 Scalability anaysis: Traces using different number of processors 19

1 Introduction

2 Anlysis with *Tareador*

In order to study which is the best approach that we should use to parallelize the code (using the *OpenMP* directives), a study about the task dependencies is needed.

In the following subsections, we will analyse the functions *merge* and *multisort* of the given code using *Tareador* to see the potential parallelism they have and also their dependences. As both functions are recursive, it is very likely that we will be able to parallelise them.

The modified code can be found in the *multisort-tareador.c* file, inside the codes directory.

2.1 Merge Function

The merge function is responsible for sorting two vectors, merging them into one. This is done using recursive calls that divide the vectors into smaller ones.

Our study in this function focuses on the two recursive calls done in the else fragment of the function. A task is created every time we make a recursive call to the function. As a consequence, some created tasks will also execute the basicmerge function.

To do that, we have used the *tareador_start_task* and *tareador_end_task* functions. The resulting code is shown below.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start,
           long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        tareador_start_task("me1");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("me1");

        tareador_start_task("me2");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("me2");
    }
}
```

Figure 1: Modified version of the merge function.

2.2 Multisort Function

The multisort function divides the input vector into 4 smaller vectors, apply a recursive call for each of them and then merges the 4 vectors into a big sorted one.

We will use the same strategy than in the merge function. This is, creating a task for each recursive call and for each call to the merge function.

We have also used the *tareador_start_task* and *tareador_end_task* functions. The code is the following:

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        tareador_start_task("mu1");
        multisort(n/4L, &data[0], &tmp[0]);
        tareador_end_task("mu1");

        tareador_start_task("mu2");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("mu2");

        tareador_start_task("mu3");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("mu3");

        tareador_start_task("mu4");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("mu4");

        tareador_start_task("me1");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("me1");

        tareador_start_task("me2");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("me2");


        tareador_start_task("me3");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("me3");

    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 2: Modified version of the multisort function.

2.3 Task dependency graph (TDG)

We can clearly see that Figure **??** is divided into two parts: the multisort and the merge.

As we said before, in the multisort function we divide the input vector into 4 smaller vectors, until the vector size is smaller than $MIN\_SORT\_SIZE \times 4L$. This can be seen at the top of the figure, as we can see 4 different boxes (green, red, yellow and purple). Moreover, there is no data sharing between them, so we can execute them at the same time.

Then, there is a new recursion level inside each box. These four calls reach the limit value, so they do a *basicmerge* call. Afterwards, we need the first two childs to terminate before executing the first merge call (and the same for the third and fourth childs with the second merge call). Thus, there exist a dependence between the recursive calls to multisort and the calls to the first two merge functions. Finally, the third merge call can be executed only when the previous two merge calls have terminated (another dependence).

On the other hand, in the merge function, we create tasks when the number of elements of the vector that we want to sort is bigger or equal than $MIN\_MERGE\_SIZE \times 2L$. We see in the figure that the two created tasks do not have any dependence between them. A *basicmerge* call will be done when we reach the limit value.

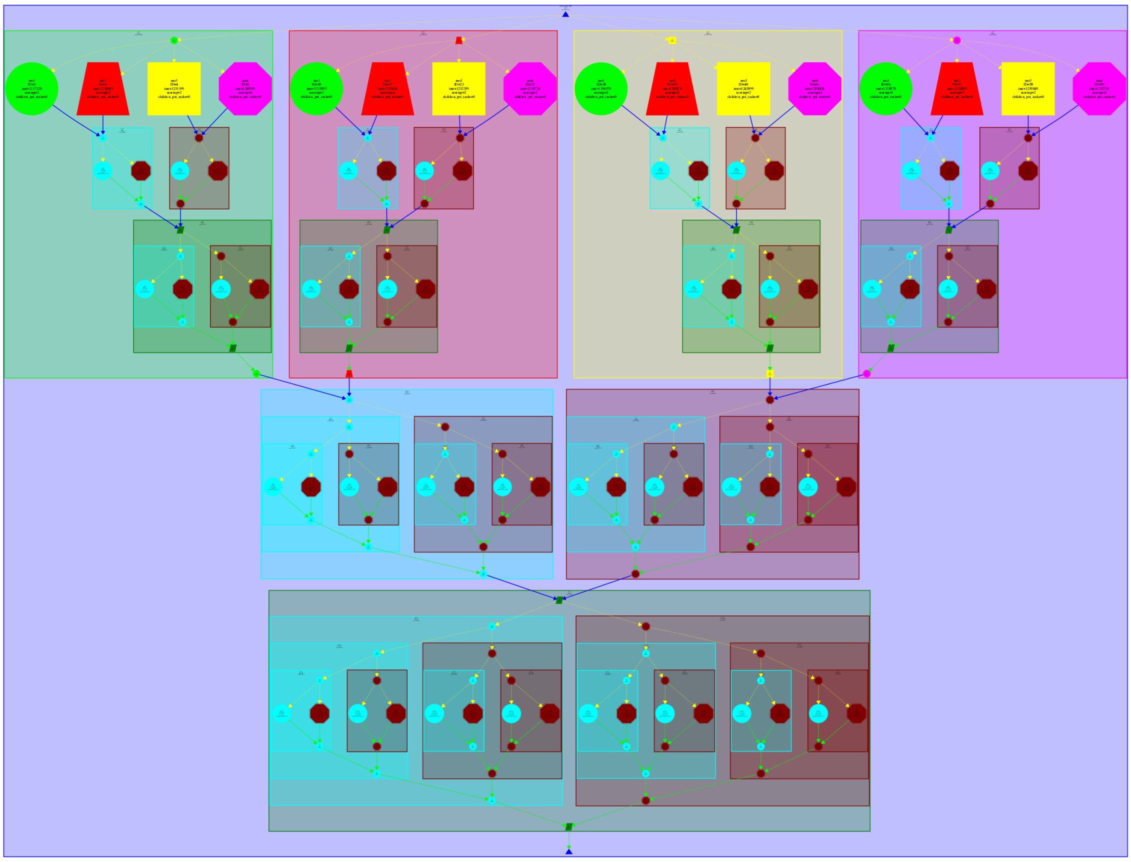

Figure 3: Task dependency graph obtained using *Tareador*.

With all these information, we can conclude that we can use a **Tree strategy** (generate a task for each recursive function) because there are no dependences between sibling tasks. Besides, we can also use a **Leaf strategy** (generate a task in each base case of the functions).

Furthermore, we must be careful about the synchronization we need between the multisort tasks and the merge ones because, as we said before, the first merge call depends on the first two multisort calls, the second merge call depends on the third and fourth multisort calls and the third merge call depends on the previous two merge calls. Without synchronization we would have wrong results due to data racing.

2.4 Parallel Performance and Scalability

The following table shows the execution time and speed-up values obtained when varying the number of processors.

Processors	Execution Time [ns]	Speed-Up
1	20334411001	1
2	10173716001	1.99872013323365
4	5086725001	3.99754478510288
8	2550595001	7.97241858979085
16	1289922001	15.7640624667507
32	1289909001	15.7642213406029
64	1289909001	15.7642213406029

Table 1: Execution Time and Speed-Up varying the number of processors.

We can see that until 16 processors, the results obtained are pretty close to the ideal ones. However, from 16 to infinite processors there is no improvement on the speed-up values. This is because in Figure **??** we see that the maximum number of tasks executed at the same time are 16. For this reason, using more than 16 processors will not have any impact on the execution time and speed-up.

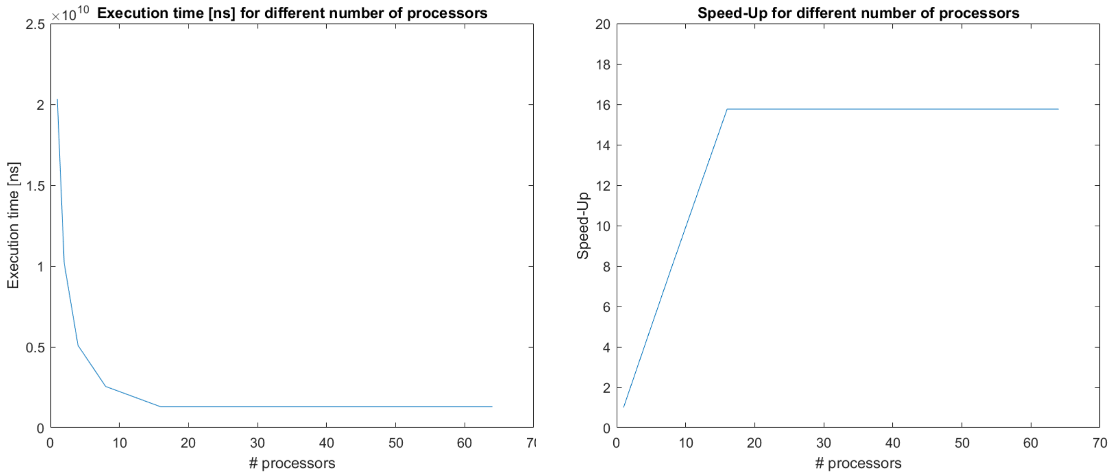

Figure 4: Execution Time and Speed-Up plots varying the number of processors.

Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear.

Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

3 Parallelization and performance analysis with tasks

As we said in the previous section, we concluded that we can use both Tree and Leaf strategies for the parallelization of the multisort and merge functions.

In this section we are going to analyse the parallelization and performance of the 2 strategies implemented using the *OpenMP* clauses. The different versions of the code can be found in the next subsections. Nevertheless, the following code will be the same for the implementation of both strategies:

```
#pragma omp parallel
#pragma omp single
multisort(N, data, tmp);
```

Figure 5: Modified fragment of the given code in the main function.

3.1 Leaf Strategy

In this strategy, we define a task for the invocations of basicsort and basicmerge functions. That is, we create a task each time we are in the base case of multisort and merge functions.

In order to do it, we have used the *#pragma omp task* clause. As we have seen in the section 2.3, there exists dependencies between some function calls, so we will need an *OpenMP* clause to synchronize these functions (*#pragma omp taskwait*).

The code can be found in the *multisort-omp-leaf.c* file inside the codes directory. However, the important changes of the code are shown below.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start,
           long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        #pragma omp task
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        merge(n, left, right, result, start, length/2);
        merge(n, left, right, result, start + length/2, length/2);
    }
}
```

Figure 6: Merge function implementing the Leaf strategy.

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        multisort(n/4L, &data[0], &tmp[0]);
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

        #pragma omp taskwait

        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

        #pragma omp taskwait

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        #pragma omp task
        basicsort(n, data);
    }
}
```

Figure 7: Multisort function implementing the Leaf strategy.

We can see in Figure **??** that only thread 0 is the one that creates tasks (in the base case of the functions), while the other tasks are executing them. Moreover, we can see that thread 0 is also executing tasks. We think that this happens because it does not create tasks very usually, so it has time to execute some of the tasks inside the pool.

On the other hand, we can also see the effect of the *#pragma omp taskwait* clause if we look at the first trace of the figure. There are plenty of red parts (synchronization parts) in all threads.

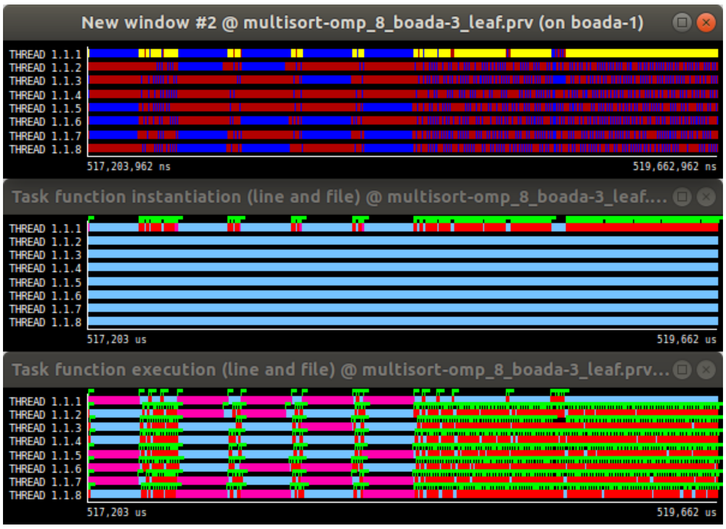

Figure 8: Fragment of the execution flow of the code using the Leaf strategy.

As in the Leaf stratgy only one thread executes the recursive part of the functions, there is a big amount of time where the other threads are doing nothing. Consequently, the speed-up plots for the complete application and only the multisort function cannot be good.

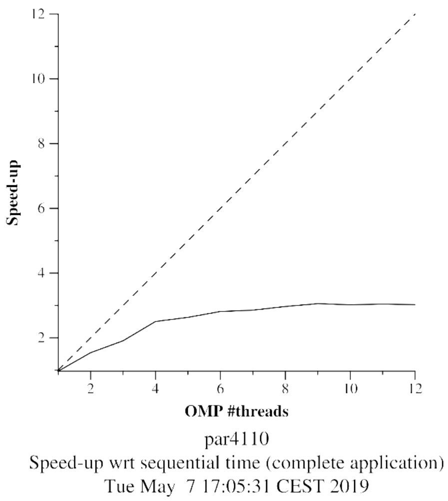

Figure 9: Speed-up plot of the complete application using the Leaf strategy.

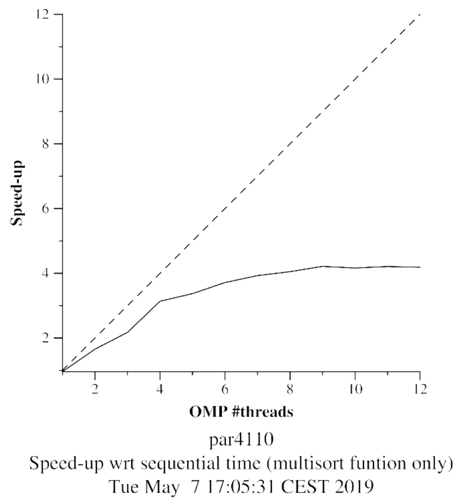

Figure 10: Speed-up plot of the multisort function using the Leaf strategy.

If we continue adding more threads to execute the code, this will result into an irrelevant improvement of both speed-up plots because there will be more threads that are doing no work, which means that the total amount of time in the Idle state will increase.

3.2 Tree Strategy

In this other strategy, we define a task during the recursive decomposition, when invoking multisort and merge recursive calls.

To do it, we have also used the *#pragma omp task* clause. However, we cannot use *#pragma omp taskwait* in the multisort function anymore because we have to stop the flow of the code until a group of recursive calls which some are siblings and others are not terminate. To achieve this, we have used the *#pragma omp taskgroup* clause. We have done 3 groups taking into account the dependences in Figure **??**: multisort calls, first two merge calls and the third merge call.

The following code can be found in the *multisort-omp-tree.c* file inside the codes directory.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start,
           long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        #pragma omp task
        merge(n, left, right, result, start, length/2);

        #pragma omp task
        merge(n, left, right, result, start + length/2, length/2);
    }
}
```

Figure 11: Merge function implementing the Tree strategy.

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp taskgroup
        {
            #pragma omp task
            multisort(n/4L, &data[0], &tmp[0]);

            #pragma omp task
            multisort(n/4L, &data[n/4L], &tmp[n/4L]);

            #pragma omp task
            multisort(n/4L, &data[n/2L], &tmp[n/2L]);

            #pragma omp task
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        }

        #pragma omp taskgroup
        {
            #pragma omp task
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);

            #pragma omp task
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L],
                  0, n/2L);
        }

        #pragma omp task
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 12: Multisort function implementing the Tree strategy.

Now, with this new strategy, all threads can execute and create tasks. At he beginning only thread 0 could create tasks, but each time it creates one, the number of threads that can create tasks increases. As a consequence, the execution time will be reduced a lot because now the total amount of time in the Idle state is reduced.

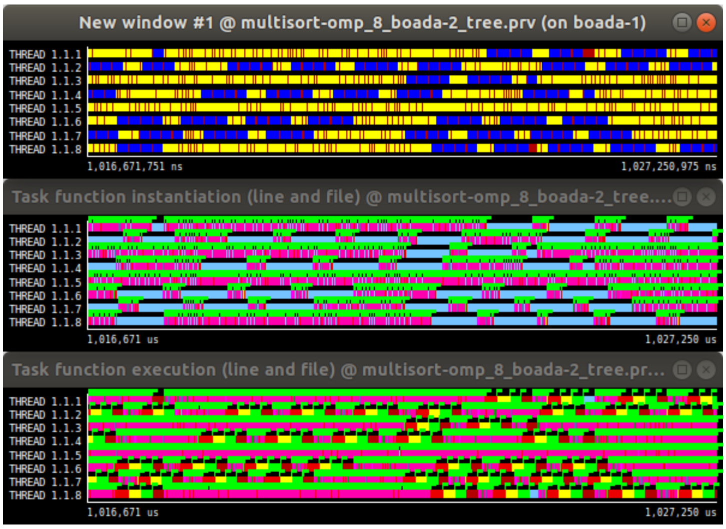

Figure 13: Fragment of the execution flow of the code using the Tree strategy.

As we have said before, the execution time is reduced a lot, improving the speed-up plots of both complete application and multisort function only. Hence, both plots will be much better than the ones obtained using the Leaf strategy.

Besides, we can see that this improvement has had a much impact on the multisort function only plot than in the complete application one.

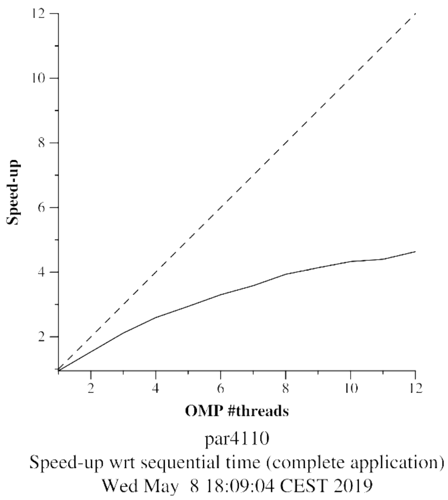

Figure 14: Speed-up plot of the complete application using the Tree strategy.

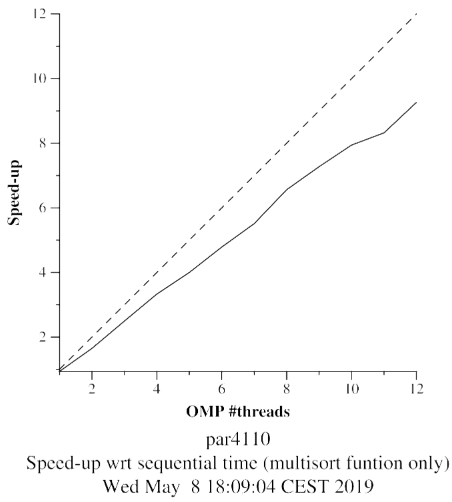

Figure 15: Speed-up plot of the multisort function using the Tree strategy.

If we continue adding more threads to execute the code, this will result into a better performance of the code. We would have better execution time and speed-up results.

3.3 Task Cut–Off Mechanism

As we have seen, in the Tree strategy there is not a maximum recursion level, so if we have a large vector input we can be calling functions recursively for a long time.

To solve this problem, we can set a maximum recursion level using a cut-off mechanism that controls it for task generation (and their granularity).

To do this, we have used the *final(condition)* clause. When the condition is true, the task has the attribute final to true. The *omp_in_final()* function uses this attribute to decide whether it can create more tasks or not.

This code can be found in the *multisort-omp-tree-cutoff.c* file inside the codes directory.

```
int CUTOFF;

void merge(long n, T left[n], T right[n], T result[n*2], long start,
           long length, int depth) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        if (!omp_in_final()) {
            #pragma omp task final(depth >= CUTOFF)
            merge(n, left, right, result, start, length/2, depth + 1);

            #pragma omp task final(depth >= CUTOFF)
            merge(n, left, right, result, start + length/2, length/2,
                  depth + 1);
        } else {
            merge(n, left, right, result, start, length/2, depth + 1);
            merge(n, left, right, result, start + length/2, length/2,
                  depth + 1);
        }
    }
}
```

Figure 16: Merge function implementing the Tree strategy using the cut-off mechanism.

```c
void multisort(long n, T data[n], T tmp[n], int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition

        if (!omp_in_final()) {
            #pragma omp taskgroup
            {
                #pragma omp task final(depth >= CUTOFF)
                multisort(n/4L, &data[0], &tmp[0], depth + 1);

                #pragma omp task final(depth >= CUTOFF)
                multisort(n/4L, &data[n/4L], &tmp[n/4L], depth + 1);

                #pragma omp task final(depth >= CUTOFF)
                multisort(n/4L, &data[n/2L], &tmp[n/2L], depth + 1);

                #pragma omp task final(depth >= CUTOFF)
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],
                          depth + 1);
            }

            #pragma omp taskgroup
            {
                #pragma omp task final(depth >= CUTOFF)
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,
                      depth + 1);

                #pragma omp task final(depth >= CUTOFF)
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L],
                      0, n/2L, depth + 1);
            }

            #pragma omp task final(depth >= CUTOFF)
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,
                  depth + 1);
        } else {
            multisort(n/4L, &data[0], &tmp[0], depth + 1);
            multisort(n/4L, &data[n/4L], &tmp[n/4L], depth + 1);
            multisort(n/4L, &data[n/2L], &tmp[n/2L], depth + 1);
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth + 1);

            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,
                  depth + 1);
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0,
                  n/2L, depth + 1);

            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,
                  depth + 1);
        }
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 17: Multisort function implementing the Tree strategy using the cut-off mechanism.

The CUTOFF variable can be set using the **-c cutoff** argument in the command line. If we do this, CUTOFF will be the maximum recursion level permitted.

To understand the cut-off mechanism, we have used a CUTOFF value of 0 and the *Paraver* tool to see the execution flow. The results are shown below.

Figure 18: Fragment of the execution flow of the code using the Tree strategy and CUTOFF = 0.

As the maximum recursion level is 0, we only create one task for each call function inside multisort. These new tasks will not be able to create tasks, and will execute te code in sequential mode.

In the recursive part, the multisort function does 7 calls (4 for the multisort and 3 for the merge). We can see this 7 created tasks in the third trace in Figure **??**. The four large executions correspond to the multisort functions. Once the first two terminate, the first merge call (yellow one) can be executed. The same happens with the second merge call (red one) when the other two multisort functions end. Finally, the third merge function can start when the other two merge calls have terminated. This happens because of the taskgroup clause.

Once we have understood the cut-off mechanism, we are going to explore the performance of the code when using different values for the maximum recursion level. To do this, we have used the **submit-cutoff-omp.sh** script.

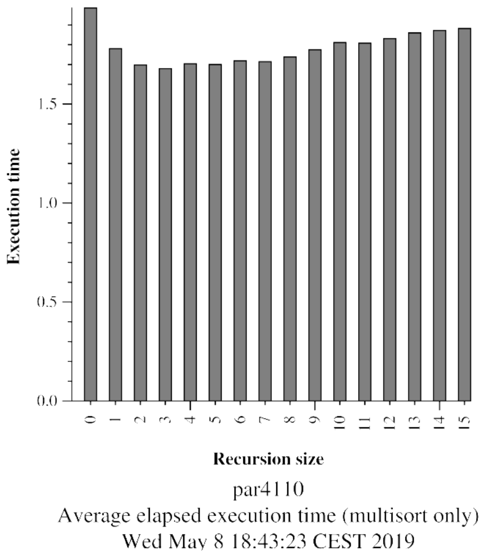

Figure 19: Execution time of the code when varying the recursion size.

With a recursion size value of 3 we get the best performance of the program. Analysing the scalability of the program using this optimum value, we see that the speed-up of the multisort function only is the best in comparison with the other two speed-up plots (Leaf and Tree strategies). However, the difference with the plot of the Tree strategy without the cut-off mechanism is very small.

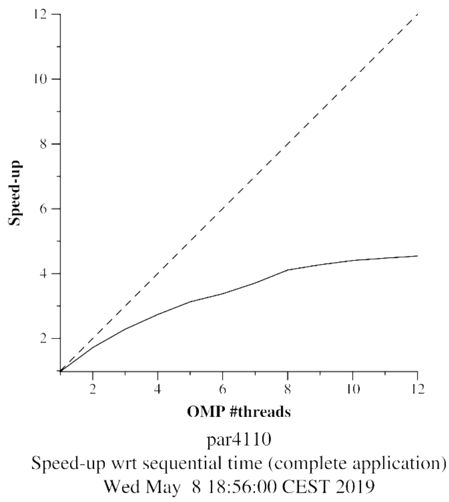

Figure 20: Speed-up plot of the complete application using the Tree strategy and the cut-off mechanism.

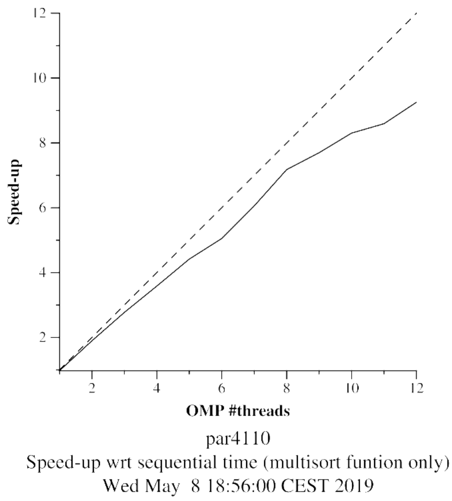

Figure 21: Speed-up plot of the multisort function using the Tree strategy and the cut-off mechanism.

1. Include the relevant portion of the codes that implement the two versions (Leaf and Tree), commenting whatever necessary.

2. For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

3. Show the changes you have done in the code in order to include a cut-off mechanism based on recursion level. Is there a value for the cut-off argument that improves the overall performance? Analyze the scalability of your parallelization with this value.

4 Parallelization and performance analysis with dependent tasks

In the previous sections we used *#pragma omp taskwait* and *#pragma omp taskgroup* to synchronize the tasks. However, it may happen that the execution time of one task is much greater than the other ones. As a result, we will loose time waiting that task to terminate.

A solution to this problem is to get rid of these two *OpenMP* clauses and use the *depend* clause to express dependencies. With this new clause, tasks do not have to wait until all previous tasks end but only the tasks that create their dependencies.

The new code is in *multisort-omp-tree-depend.c* file inside the codes directory.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start,
           long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        #pragma omp task
        merge(n, left, right, result, start, length/2);

        #pragma omp task
        merge(n, left, right, result, start + length/2, length/2);

        #pragma omp taskwait
    }
}
```

Figure 22: Merge function implementing the Tree strategy using the dependence clauses.

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp task depend(out:data[0])
        multisort(n/4L, &data[0], &tmp[0]);

        #pragma omp task depend(out:data[n/4L])
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);

        #pragma omp task depend(out:data[n/2L])
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);

        #pragma omp task depend(out:data[3L*n/4L])
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);


        #pragma omp task depend(in:data[0],data[n/4L])
                         depend(out:tmp[0])
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);

        #pragma omp task depend(in:data[n/2L],data[3L*n/4L])
                         depend(out:tmp[n/2L])
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

        #pragma omp task depend(in:tmp[0],tmp[n/2L])
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

        #pragma omp taskwait
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 23: Multisort function implementing the Tree strategy using the dependence clauses.

The first merge call cannot be executed until the previous tasks with *depend(out: data[0])* and *depend(out: data[n/4L])* terminate. Now, even though the third or fourth multisort calls have not terminated, the first merge call can be executed. The same happens for the rest of the calls with the depend clause. Thus, we will reduce a bit the execution time of the program.

Nevertheless, we must use *#pragma omp taskwait* at the end of both multisort and merge functions. Without them, the sorting would not be correct because it could happen that while the last merge call inside the multisort function is being executed, we terminate this function and start with other tasks. The next task would think that the vector is already sorted altough that merge call would have not terminated yet. The same happens inside the merge function.

The most important thing we see in Figure **??** is that there is even less parts in Idle time than in the Tree flow (Figure **??**). Hence, the execution time is reduced a bit with respect to the one using the Tree strategy.

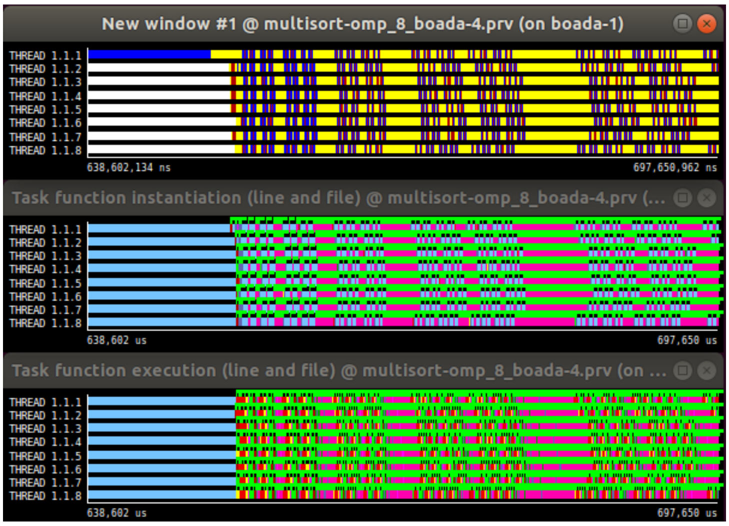

Figure 24: Fragment of the execution flow of the code using the Tree strategy and the depend clause.

Finally, in the scalability plots of this program we can observe that both have the best speed-ups with respect to all previous speed-up plots. This has been possible with the use of the depend clause.

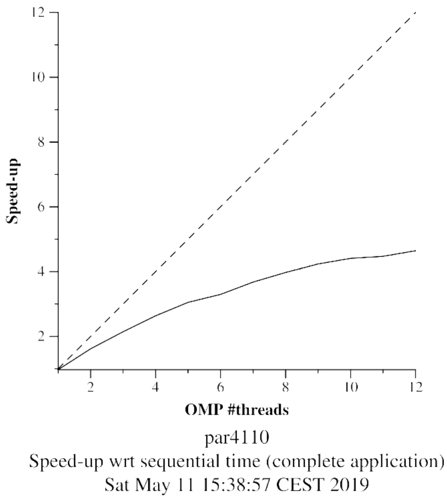

Figure 25: Speed-up plot of the complete application using the Tree strategy and the cut-off mechanism.

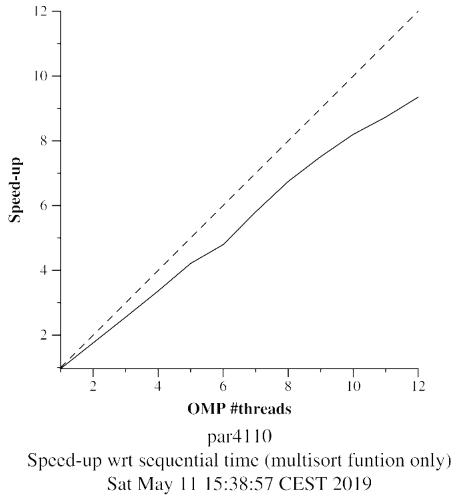

Figure 26: Speed-up plot of the multisort function using the Tree strategy and the cut-off mechanism.

1. Include the relevant portion of the code that implements the Tree version with task dependencies, commenting whatever necessary.

2. Reason about the performance that is observed, including the speed–up plots that have been obtained for different numbers of processors and with captures of Paraver windows to justify your reasoning.

5 Annex

5.1 Scalability anaysis: Traces using different number of processors

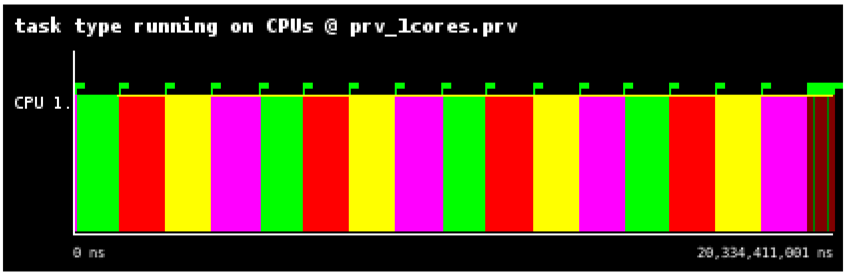

Figure 27: Execution flow of the code using 1 processor.

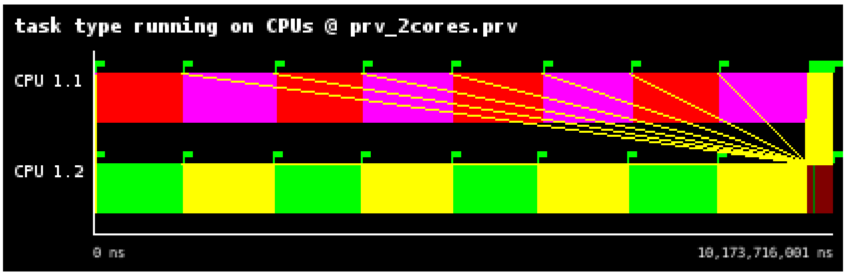

Figure 28: Execution flow of the code using 2 processors.

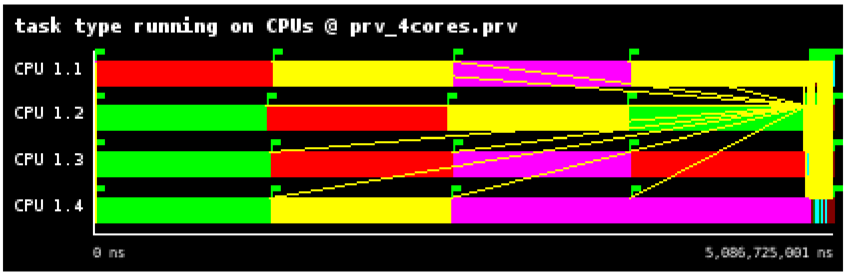

Figure 29: Execution flow of the code using 4 processors.

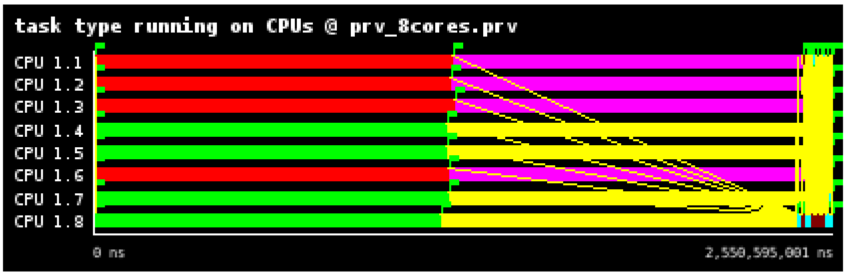

Figure 30: Execution flow of the code using 8 processors.

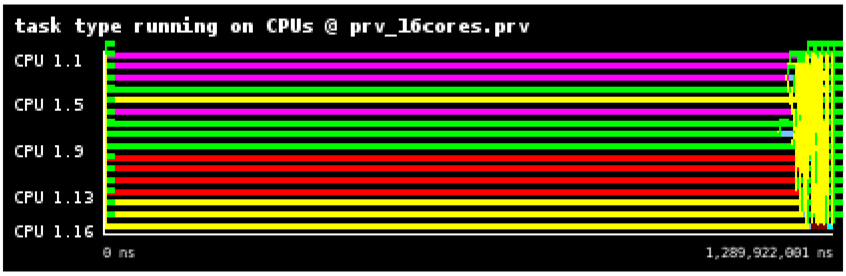

Figure 31: Execution flow of the code using 16 processors.

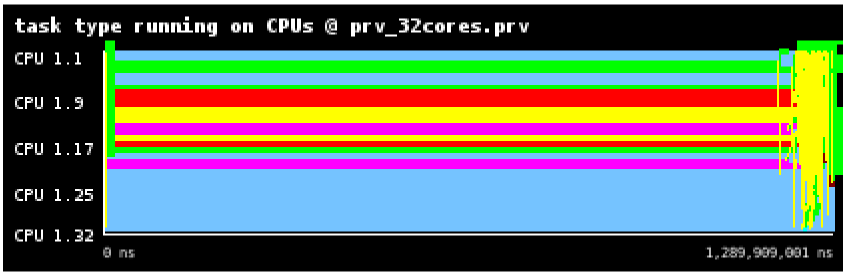

Figure 32: Execution flow of the code using 32 processors.

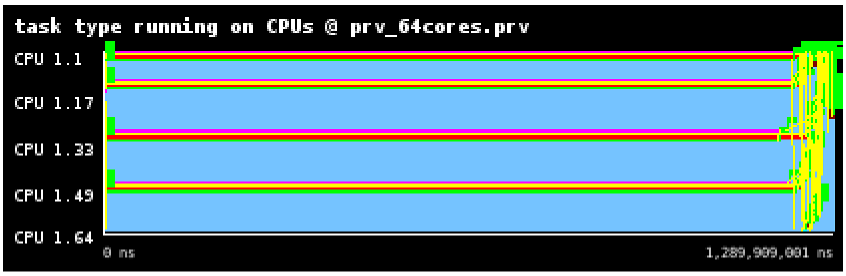

Figure 33: Execution flow of the code using 64 processors.